MINOR PROJECT REPORT

ON

IMPLEMENTATION OF DISTRIBUTED FILE SYSTEM

Specialization	SAP ID	NAME
B.Tech H CSE BigData (B1 H)	500107098	Harvijay Singh
B.Tech CSE BigData (B2 NH)	500105724	Piyush Vedwal
B.Tech CSE BigData (B2 NH)	500107565	Sparsh Agarwal

under the guidance of

Dr. DEEPAK KUMAR SHARMA

Data Science Cluster

School of Computer Science

UNIVERSITY OF PETROLEUM AND ENERGY STUDIES

Dehradun-248007

CANDIDATE'S DECLARATION

We hereby certify that the project work entitled **"Implementation of Distributed File System"** in partial fulfillment of the requirements for the award of the Degree of **BACHELOR OF TECHNOLOGY** in **COMPUTER SCIENCE AND ENGINEERING** with specialization in **Big Data**, submitted to the **Data Science Cluster, School of Computer Science, UPES, Dehradun**, is an authentic record of our work carried out during a period from **January 2025 to May 2025** under the supervision of **Dr. Deepak Kumar Sharma**, **SoCS**.
The matter presented in this project has not been submitted by us for the award of any other degree of this or any other University.

Harvijay Singh	**500107098**
Piyush Vedwal	**500105724**
Sparsh Agarwal	**500107565**

This is to certify that the above statement made by the candidates is correct to the best of my knowledge.

Dr. Deepak Kumar Sharma
Project Guide
Systematics Cluster, SoCS
Date: April 28, 2025

ACKNOWLEDGEMENT

We would like to express our heartfelt gratitude to our project guide, **Dr. Deepak Kumar Sharma**, for his expert guidance, encouragement, and unwavering support throughout the course of this project. His insights and advice were instrumental in shaping the work to its present form.

We are deeply thankful to **Dr. Virender Kadyan**, Cluster Head of the Data Science Cluster, for providing essential resources and continuous support that enabled the smooth execution of our project on the ”Implementation of Distributed File System.” We also extend our sincere thanks to the **Dean, School of Computer Science, UPES** for facilitating the necessary infrastructure for this work.

We appreciate the timely assistance and encouragement provided by our **Course Coordinator, Dr. Prabhat Ranjan Singh**, and our **Activity Coordinator, Dr. Sachin Chaudhary**, whose guidance proved invaluable during various stages of the project.

We are grateful to our friends and colleagues for their collaborative spirit, constructive feedback, and thoughtful discussions that contributed to the overall success of this endeavor.

Lastly, we owe our deepest thanks to our families for their unconditional love, patience, and encouragement, without which this academic achievement would not have been possible.

Harvijay Singh	**500107098**
Piyush Vedwal	**500105724**
Sparsh Agarwal	**500107565**

ABSTRACT

With the growing need for large-scale, high performance file storage, distributed file systems have become essential for delivering scalability, security, and efficiency. This project focuses on creating a distributed file storage and management system using Python, featuring local storage support and role-based access control (RBAC). Designed for enterprise-level use, the system ensures quick data retrieval, fault tolerance, and effective load balancing. It also tackles common challenges found in traditional file management solutions, such as limited search capabilities, weak access restrictions, and high response times, by incorporating advanced methods like data sharding, indexing, and robust RBAC.

Contents

1 Introduction

Traditional file storage systems struggle with scalability, security, and efficiency as data volume increases. No-SQL based distributed systems address some of these issues but often lack advanced querying capabilities and fine-grained access control.

1.1 Purpose of the Project

With the explosive growth in digital data, secure and efficient file storage and transfer mechanisms have become critical. This project aims to develop a robust, user-friendly system that ensures confidential, redundant, and scalable file handling across distributed servers. Through file encryption, intelligent splitting, and load-balanced storage on multiple servers, the system guarantees data integrity, fault tolerance, and security.

The primary goal is to simplify secure data management for end users by providing a Graphical User Interface (GUI) that abstracts complex backend processes such as file encryption, networking, and distributed storage.

1.2 Target Beneficiaries

This system is designed for individuals and organizations seeking a reliable and secure solution to store sensitive files across multiple storage locations. The intended beneficiaries include:

1. Small and medium-sized enterprises (SMEs) that require affordable, secure storage without depending on third-party cloud services.
2. Educational institutions aiming to provide students with hands-on experience in encryption, networking, and storage technologies.
3. Developers and researchers specializing in distributed systems or cybersecurity who need a local, sandboxed environment for testing secure file transfers.
4. Home users looking to protect personal data and create redundant backups across multiple devices or network locations.

1.3 Project Scope

The project covers the following major components:

- **File Selection and Encryption:** Users can select a file, which is then encrypted using Fernet (based on AES encryption) to maintain data confidentiality.
- **File Splitting:** After encryption, the file is divided into multiple segments to enable parallel storage and enhance fault tolerance.
- **Redundant Storage with Load Balancing:** Each segment is stored on two servers — a primary and a replica — with a basic load-balancing strategy to prevent any single server from becoming overloaded.
- **Multithreaded Server Operations:** Servers handle multiple client requests concurrently using Python threading, making the system responsive and scalable.
- **Reconstruction and Decryption:** The system supports the retrieval of file parts from either main or replica servers, decrypts them, and reconstructs the original file.

- **Interactive GUI:** A Tkinter-based interface makes the system accessible to users with minimal technical knowledge.
- **Metadata Management:** The system tracks file activity, including upload/download timestamps, source and destination paths, and encryption details in a local metadata log (`file_metadata.json`).

1.3.1 Scalable and Secure Infrastructure

The design allows the system to scale horizontally by adding more server endpoints. Each server operates independently and concurrently, enabling:

1. **Scalability:** Additional storage nodes can be integrated without major architectural changes.
2. **Fault Tolerance:** If a primary server fails, the system can fall back on replica servers for data retrieval.
3. **Data Confidentiality:** Files are encrypted before storage, ensuring they remain protected even if servers are compromised.

The system architecture is modular and designed to evolve, supporting cloud migration or integration with existing enterprise storage solutions in future iterations.

1.3.2 Interactive Reports

The GUI not only provides visual feedback on ongoing operations (e.g., file transfer progress, connection status) but also includes:

1. Animated status updates for real-time operations.
2. Transfer logs and metadata accessible in human-readable formats.
3. Visual representation of server load distribution (future scope).
4. User alerts for failed operations and successful reconstructions.

These interactive elements make the system transparent and user-centric.

1.4 Challenges Addressed

The system effectively addresses several common issues in file storage and transfer:

1. Single point of failure – resolved via redundancy.
2. Unencrypted transfer risks – mitigated using symmetric key encryption.
3. Manual intervention – minimized through automation in file splitting, transfer, and reconstruction.
4. Lack of feedback – solved using progress indicators and logs.

1.5 Technology Stack Overview

Component	**Technology Used**
Language	Python
GUI	Tkinter
Networking	Python Sockets
Encryption	Fernet (Cryptography Library)
Multithreading	`threading` module
Data Logging	JSON Metadata (`file_metadata.json`)
File Handling	OS and File I/O

2 PROBLEM STATEMENT

Traditional file storage systems face challenges with performance, security, and scalability as data grows. Slow retrieval, unoptimized search mechanisms, and inadequate access control frameworks lead to inefficiencies. Distributing files across nodes creates synchronization issues, while existing systems struggle with high request volumes and lack intelligent load balancing. This project aims to develop a secure, scalable distributed file handling system with encryption, redundancy, and load balancing, offering a user-friendly interface and enhanced performance.

3 Objectives

The main aim of this project is to develop a distributed file storage system that is both scalable and highly efficient. To achieve this, several key objectives have been identified:

1. Develop complete support for creating, reading, updating, and deleting files within the system.
2. Build robust fault-tolerance features by introducing replication and automatic failover strategies.
3. Improve search and retrieval speeds by integrating effective indexing and caching techniques.
4. Distribute data intelligently across nodes by applying sharding strategies combined with consistent hashing.
5. Design and implement a load balancing mechanism to evenly manage user requests and prevent bottlenecks.

4 SYSTEM DESIGN

The system is built around a modular and distributed architecture, carefully designed to tackle key challenges related to performance, security, and scalability in file storage and management. Each module serves a dedicated purpose, while collectively ensuring the system remains reliable, efficient, and responsive. A detailed overview of the core components is provided below:

4.1 Storage Nodes

Function: Act as the physical or logical units where encrypted file fragments are stored.

Design Subunits:

1. **Primary and Replica Servers:** Each file part is stored on two servers—a primary and a replica—ensuring redundancy and fault tolerance.

2. **Server Directories:** Each server maintains its own local storage directory for file fragments, ensuring file isolation and simplified management.

3. **Threaded Server Operations:** Servers support concurrent client connections using multithreading, enabling them to handle multiple uploads/downloads without performance degradation.

4. **Port-Configurable Instances:** Multiple servers can be launched on separate ports, simulating a distributed environment over LAN or WAN.

4.2 Metadata Management

Function: Maintains critical information about file transfers, encryption keys, server assignments, and timestamps.

Design Subunits:

1. **Metadata File (`file_metadata.json`):** Logs file name, storage nodes, part identifiers, encryption key hash, and transfer timestamps.

2. **Indexing System:** Enables easy retrieval and reconstruction by indexing which parts are stored on which servers.

3. **Audit Trail:** Supports transparency and traceability for all file operations, useful for debugging, monitoring, or compliance audits.

4.3 Access Control Module

Function: Controls access to sensitive file operations through rule-based mechanisms.

Design Subunits:

1. **Encryption Layer (Fernet AES):** Acts as the first level of access control by requiring possession of the correct encryption key for decryption.

2. **Role-Based Access Control (RBAC) (Future Scope):** The design anticipates integration of user roles (e.g., admin, editor, viewer) to grant permissions accordingly.

3. **Key Management System (Planned):** A centralized or distributed mechanism to safely manage and distribute encryption keys to authorized users.

4.4 Load Balancer

Function: Evenly distributes file parts across multiple storage nodes to prevent bottlenecks and optimize performance.

Design Subunits:

1. **Round-Robin Strategy:** Alternates server selection in a fixed order to balance load during uploads.

2. **Replica-Aware Logic:** Ensures that each file part is stored on two distinct nodes to prevent single points of failure.

3. **Scalable Routing Engine (Future Enhancement):** Potential integration with a dynamic load estimator to allocate parts based on real-time server performance metrics.

4.5 Query Optimization Module

Function: Improves retrieval speed and responsiveness when reconstructing or accessing stored files.

Design Subunits:

1. **Parallel Retrieval Mechanism:** Fetches multiple file parts concurrently using multithreaded download operations.

2. **Pre-indexed Metadata Access:** Reduces time to identify storage locations by consulting prelogged metadata.

3. **Cache Layer (Future Scope):** Potential implementation of a lightweight caching mechanism for recently accessed files or parts.

4.6 Replication & Fault Tolerance

Purpose: This module guarantees uninterrupted access to data, even when servers encounter failures or file corruption.

Key Components:

1. **Redundant Storage:** Every file segment is duplicated and stored across two different servers, ensuring that a backup is always available if one server goes offline.

2. **Server Health Independence:** The system is capable of reconstructing complete files as long as at least one intact copy of each segment can be accessed.

3. **Automatic Fallback:** During file retrieval, if the primary server is unresponsive, the system seamlessly switches to the backup server to fetch the necessary parts.

4. **Real-Time Error Notifications:** Users are promptly alerted through the graphical interface if any part of a file is missing, corrupted, or otherwise inaccessible, helping maintain system transparency and trust.

5 Area of Application

The secure file splitting and distributed storage system offers versatile applications across multiple industries and sectors where data security, high availability, and scalability are critical. Thanks to its modular design and intuitive interface, the system is well-suited for both large-scale enterprise implementations and educational use. A detailed overview of its primary application areas is presented below:

5.1 Enterprise Document Management

Large enterprises and corporations frequently handle massive volumes of sensitive documents, including legal agreements, financial statements, and internal reports. The proposed system addresses these needs by offering:

1. End-to-end encryption to safeguard corporate documents from unauthorized access.
2. Role-based access control (planned) to define who can view, edit, or retrieve sensitive data.
3. Redundancy and fault tolerance, reducing the risk of data loss from hardware failures.

Use Case Example: Multinational corporations needing to distribute backup archives across multiple data centers.

5.2 Cloud Storage Systems

Cloud storage providers and users benefit from systems that are both scalable and secure. By simulating a distributed cloud model with local servers, this project demonstrates:

1. Scalable storage models through load-balanced data distribution.
2. Secure data fragmentation, which adds a layer of protection compared to traditional monolithic file uploads.
3. Simplified integration potential with cloud providers like AWS, Azure, or private cloud networks.

Use Case Example: Custom-built private cloud solutions for startups or research labs.

5.3 Healthcare & Finance

Industries such as healthcare and finance handle highly sensitive, regulated data, making security and traceability critical.

1. Encrypted file handling ensures compliance with data protection laws (HIPAA, GDPR, etc.).
2. Detailed metadata tracking provides an audit trail, which is essential for regulatory reporting and investigations.
3. Fault-tolerant architecture helps preserve critical medical records or transaction logs even in case of partial system failure.

Use Case Example: Hospitals storing patient diagnostic records or banks archiving transaction history securely.

5.4 Remote Collaboration

With the rise of globally distributed teams, secure data sharing across physical and digital borders has become a necessity.

1. GUI-based access enables non-technical users to share and retrieve files securely.
2. Redundant, distributed storage allows collaborative access without relying on centralized points of failure.
3. Metadata logs and version tracking (future enhancement) could enable version history or collaborative editing verification.

Use Case Example: International research teams securely sharing datasets or reports.

5.5 Big Data Storage & Analysis

Data-intensive domains require efficient systems to store, transfer, and process large files—often terabytes in size.

1. Splitting large files into smaller parts facilitates faster, parallel data ingestion and retrieval.
2. Redundancy and concurrent transfers optimize I/O operations in distributed analytics systems.
3. Scalable server infrastructure allows for increased throughput as data demands grow.

Use Case Example: Data science platforms storing raw input data and intermediate processing outputs.

5.6 Legal & Compliance Services

Law firms and regulatory bodies handle confidential, case-sensitive documents that require secure handling and long-term storage.

1. Encryption and replication protect client data and legal archives from leaks or loss.
2. Detailed logs and timestamps ensure evidence of handling, essential for audit and chain-of-custody requirements.

Use Case Example: Law firms archiving deposition files across multiple secure locations.

5.7 Educational Institutions & Training Platforms

The system can also be used as a learning tool to teach students about distributed systems, cybersecurity, and file system design.

1. Hands-on simulation of real-world file storage environments.
2. Understandable GUI interface for students to interact with abstract technical concepts.
3. Modular architecture allows for customization and experimentation in lab environments.

Use Case Example: Universities conducting labs on distributed computing, networking, or cryptography.

6 Methodology

6.1 Planning & Requirement Analysis

1. **Functional Requirements**

   The system allows users to select files through an intuitive graphical interface, supports customizable file splitting either by percentage or fixed size, manages secure storage and retrieval of file fragments, and ensures complete restoration of original files. These essential functions are designed to meet operational needs while safeguarding data integrity throughout every stage.

2. **Non-Functional Requirements**

   Built using Python and PyQt5 for cross-platform compatibility, the system is engineered for fault tolerance and minimal downtime. The architecture emphasizes UI responsiveness even during intensive file operations, while advanced versions incorporate session-based or key-based validation for enhanced security. The system is benchmarked to handle multi-gigabyte files efficiently and is designed for horizontal scalability through the addition of new servers.

3. **Security and Scalability Benchmarks**

   The design embeds access control mechanisms to prevent unauthorized access while maintaining the ability to scale seamlessly. Security protocols are rigorously applied, and the system's performance is validated against standards ensuring its capacity to handle large datasets and growing infrastructure without a decline in service quality.

6.2 Design & Architecture Development

1. **System Architecture**

   The system adopts a modular structure featuring a client-side GUI for user interaction, a core engine (s1.py, s2.py) that manages file operations and metadata, and an optional server simulation module for testing distributed storage. This separation of responsibilities promotes easier development and testing of individual components.

2. **Data Flow Model**

   File operations follow a structured pipeline: users select files through the GUI, which are then processed by the core engine for splitting and storage, with comprehensive metadata generation. Reconstruction operations reverse this flow, using metadata to locate and validate file parts before seamless reassembly.

3. **Partitioning & Load Balancing Design**

   The partitioning logic adapts dynamically to files of varying sizes, with future-proofing for integrating advanced load balancing methods. The design anticipates incorporating server monitoring and round-robin scheduling to optimize resource distribution across a distributed environment.

6.3 Implementation

1. **File Management Module**

   Developed using Python's standard libraries (os, shutil) along with custom-built extensions, this module oversees all file operations such as format conversion, file splitting, and error handling. Strong exception handling ensures stability against permission issues, limited disk space, and file corruption incidents.

2. **Data Partitioning & Sharding Mechanism**

   The system offers flexible options for partitioning files, allowing users to define splits by percentage or specific sizes. Sharding demonstrations simulate distributed storage principles by allocating file parts across various logical locations, laying the foundation for real-world distributed deployment.

3. **Metadata & Reconstruction Logic**

   A comprehensive metadata system records critical information like part identifiers, sizes, and file attributes. The reconstruction process leverages this metadata to ensure accurate file reassembly, guaranteeing that the restored file matches the original bit-for-bit.

6.4 Testing

1. **Unit Testing**

   Comprehensive testing validates all standalone modules, from file selection handlers to partition generators. Test cases address normal conditions as well as edge cases, such as minimal file sizes, pre-processed inputs, and invalid user operations, ensuring a resilient system.

2. **Integration Testing**

   Full-system tests combine GUI and core engine operations, verifying end-to-end workflows. Checksum validation confirms file integrity post-reconstruction, while stress testing assesses the system's performance under concurrent operations involving diverse file types and sizes.

3. **Performance Simulation**

   Benchmark tests are conducted across a range of file sizes—from 100MB to several gigabytes—while monitoring system memory and processing metrics. These simulations identify potential bottlenecks and provide insights for optimizing memory management and processing speed.

6.5 Deployment

1. **Containerization with Docker**

   Docker containers ensure consistent environments across different platforms, with volume mounts facilitating easy access to the host filesystem. This approach simplifies dependency management and supports seamless transitions across development, testing, and production stages.

2. **Distributed Environment Deployment**

   Distributed capabilities are validated through multi-node simulations using either virtual machines or container clusters. These deployments rigorously test part distribution algorithms, cross-node restoration reliability, and system resilience under realistic network and load conditions.

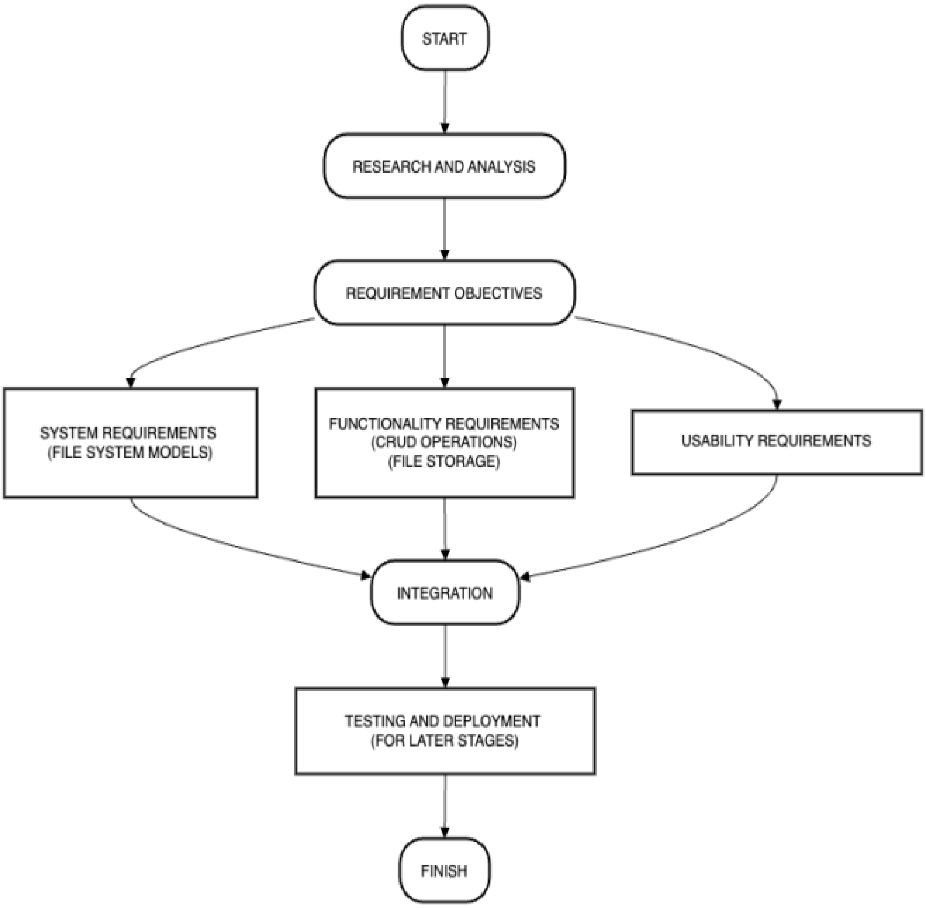


Figure 1: Methodology Workflow

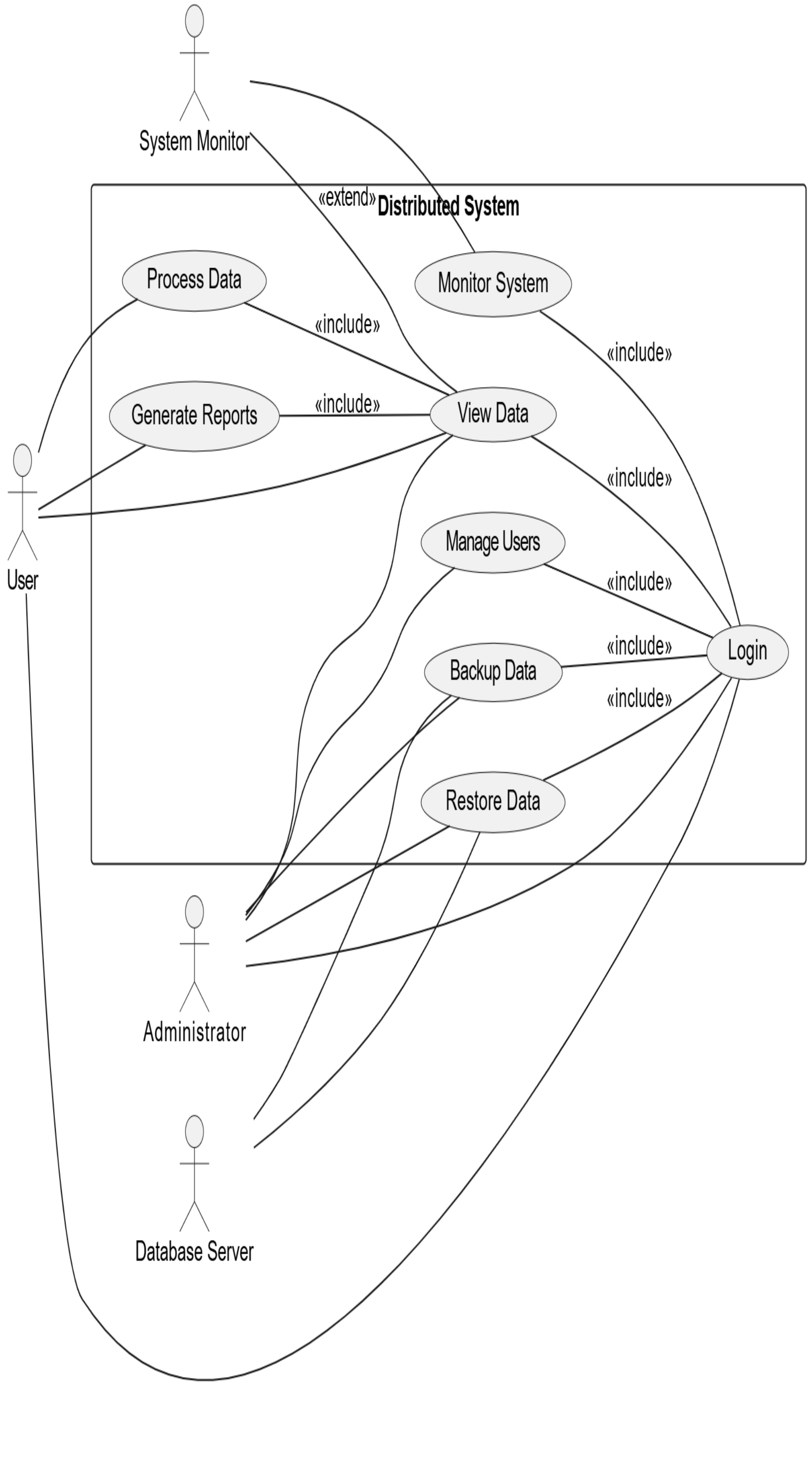


Figure 2: Use Case Diagram

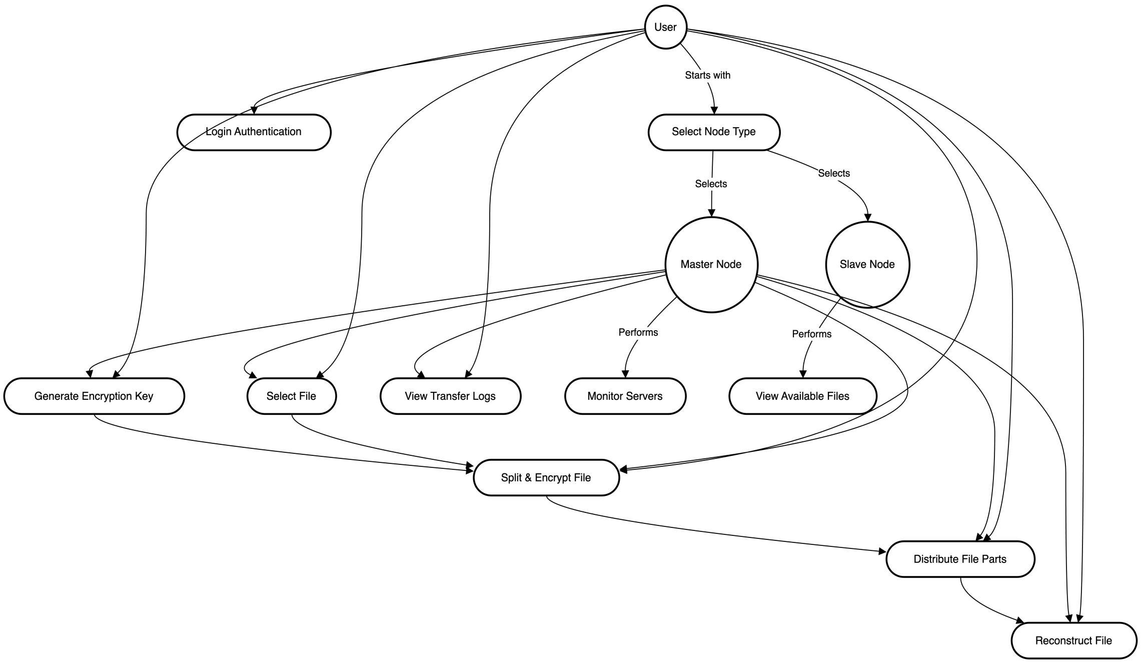


Figure 3: System Workflow

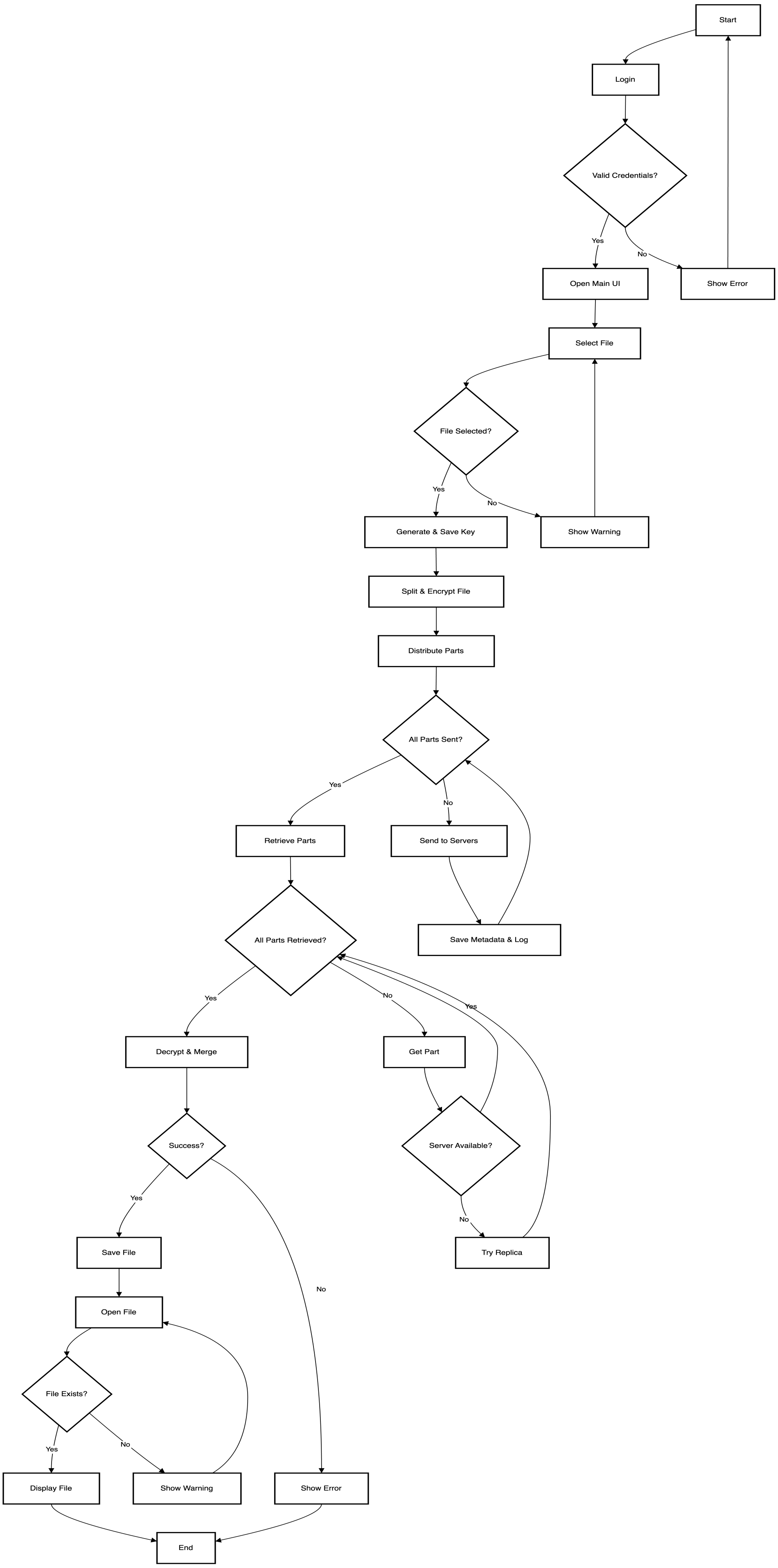


Figure 4: Activity Diagram

6.6 File Storage & Retrieval Algorithm

1. Implements data replication to ensure fault tolerance.
2. Utilizes caching techniques for faster access to frequently requested files.

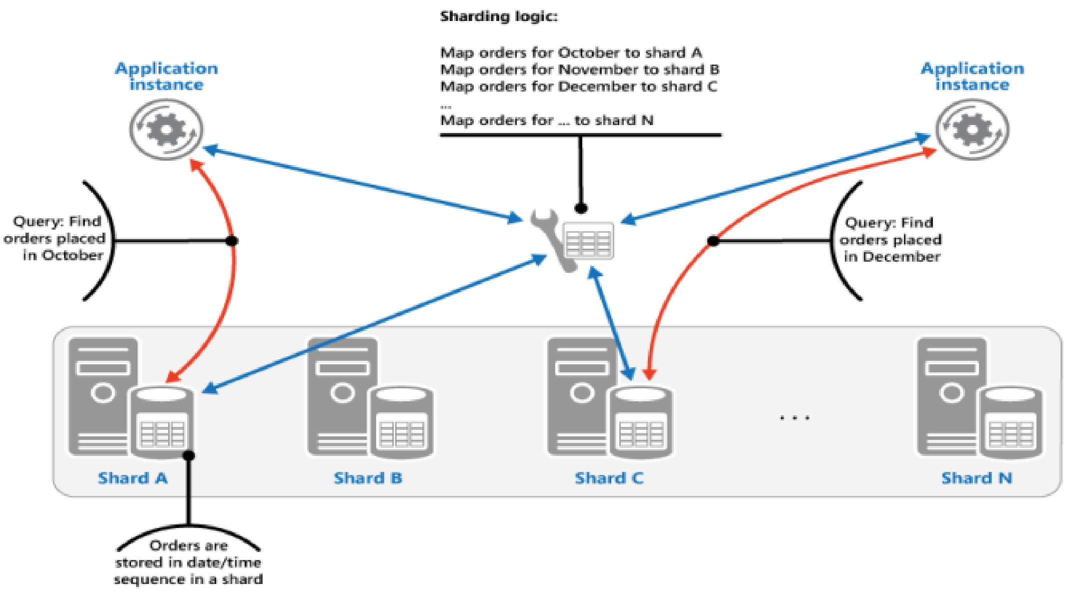

Sharding logic:
Map orders for October to shard A
Map orders for November to shard B
Map orders for December to shard C
...
Map orders for ... to shard N
Application instance
Application instance
Query: Find orders placed in October
Query: Find orders placed in December
Shard A
Shard B
Shard C
...
Shard N
Orders are stored in date/time sequence in a shard

Figure 5: Distributed Data Storage

6.7 Sharding & Data Distribution Algorithm

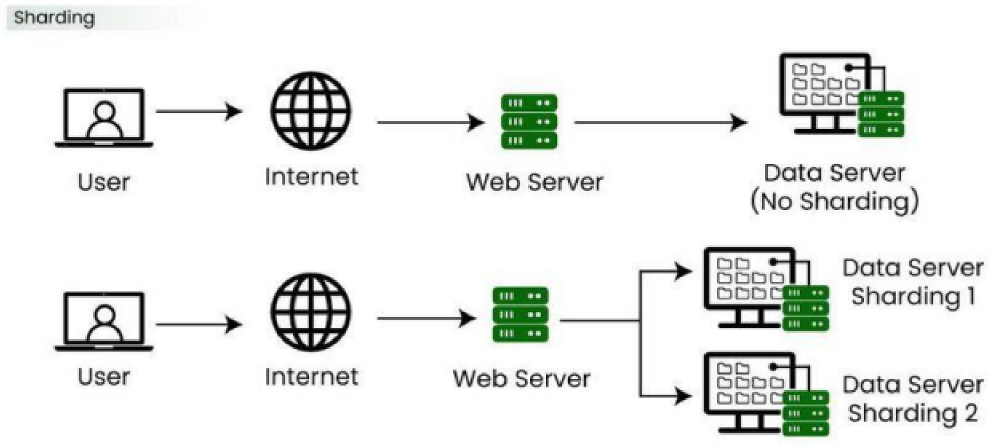


Figure 6: Data Sharding and Storage

1. Employs consistent hashing for flexible and dynamic data partitioning.
2. Distributes workload evenly across storage nodes to avoid bottlenecks.

6.8 Role-Based Access Control Algorithm

1. Authenticates users and assigns specific roles.

2. Verifies permissions before granting access to resources.

3. Enforces a hierarchical structure for access control management.

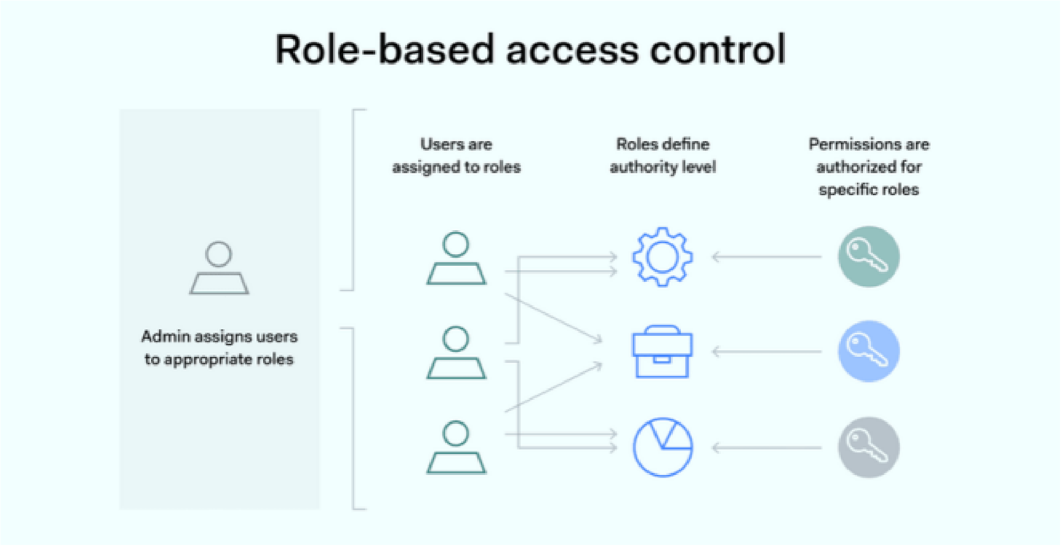

Role-based access control
Admin assigns users to appropriate roles
Users are assigned to roles
Roles define authority level
Permissions are authorized for specific roles

Figure 7: Access Control Structure

7 SWOT Analysis

A SWOT analysis is a strategic planning tool that evaluates the Strengths, Weaknesses, Opportunities, and Threats related to a system. For the secure file storage and splitting system, this analysis provides an understanding of its current capabilities, limitations, potential areas of growth, and external risks.

7.1 Strengths

These are internal attributes that give the system a competitive edge.

1. **Security-First Architecture**

   The system employs AES-based encryption to ensure data confidentiality and integrity, while storing file fragments across multiple servers to minimize the impact of potential breaches by containing exposure to isolated segments.

2. **Fault Tolerance and Redundancy**

   Through dual-server replication, the system maintains data recoverability during single-node failures, significantly enhancing overall system reliability and resilience against hardware or network disruptions.

3. **Scalable Modular Design**

   The architecture supports seamless addition of storage nodes for infrastructure expansion, with independent and upgradable modules (load balancing, metadata management, access control) that allow for targeted improvements without system-wide modifications.

4. **User-Friendly Interface**

   Built with Tkinter, the GUI enables non-technical users to interact effectively with the system, supported by comprehensive progress tracking, detailed logging, and intuitive error notifications that collectively enhance the user experience.

5. **Detailed Metadata Logging**

   The system provides transparent tracking of file operations through extensive metadata recording, facilitating debugging processes, compliance auditing, and potential forensic investigations when required.

Manual Server Setup Server deployment requires manual configuration of ports and paths, without automated server discovery or dynamic scaling, making infrastructure management less flexible.

No Cloud or Web Access The system operates only within LAN or localhost environments and does not integrate with cloud storage services or provide web-based client access, limiting broader deployment.

Basic Key Management Encryption key management is left entirely to users, with no centralized safeguards, increasing the risk of key loss or mismanagement that could lock users out of their data.

Limited Role-Based Permissions Although role-based access control (RBAC) is planned, it is not yet fully implemented, restricting support for complex permission structures across multiple user groups.

7.2 Threats

These are potential external challenges or risks that may impact the system's success.

1. **Security Breaches from Human Error**

   The system remains vulnerable to security compromises resulting from user mishandling of encryption keys or server credentials, which could lead to permanent data loss or unauthorized access despite robust technical safeguards.

2. **Technological Obsolescence**

   Rapid evolution in storage technologies and security standards may render aspects of the current design outdated, requiring continuous updates to maintain compatibility and competitive performance.

3. **Compliance and Legal Challenges**

   Increasingly stringent data privacy regulations (GDPR, HIPAA, etc.) may necessitate additional controls and features not currently implemented, potentially requiring significant architectural changes to achieve compliance.

4. **Scalability Under Extreme Load**

   The absence of horizontal scaling mechanisms and containerized deployment options may limit performance during peak usage periods, particularly as user bases grow and file volumes increase exponentially.

5. **Competition from Established Platforms**

   Mature commercial solutions like Dropbox, Google Drive, and OneDrive continue to advance their feature sets, presenting significant challenges in differentiating this system's value proposition to potential users.

8 LIMITATIONS

1. **Manual Server Initialization**
   (a) Servers (`s1.py`, `s2.py`) need to be started manually on different ports.
   (b) No central controller exists to orchestrate server health, discovery, or dynamic node addition.

2. **No User Authentication or Login System**
   (a) The application (`a_ui.py`) lacks user identity management.
   (b) Anyone with access to the GUI and encryption key can decrypt and retrieve files, limiting secure multi-user usage.

3. **Static Port Configuration**
   (a) Server addresses and ports are hardcoded and need manual entry in the UI.
   (b) No dynamic discovery of active storage nodes or fallback servers beyond the defined ones.

4. **No Real-Time Key Management**
   (a) Users must handle encryption keys manually.
   (b) If the key is lost, the file cannot be decrypted, with no backup or recovery system in place.

5. **No Support for Large File Optimization**
   (a) The file splitting logic (`a_ui.py`) simply splits files into two parts.
   (b) There's no configurable chunk size or adaptive partitioning for very large files.

6. **Absence of Multi-threaded Server Health Checks**
   (a) While servers are multi-threaded to handle multiple client connections, there's no built-in health monitor to check if a node has gone down.

7. **Basic Load Distribution Strategy**
   (a) A simplistic round-robin mechanism is used to assign servers.
   (b) No intelligent load balancing or performance-based server selection exists.

8. **UI Lacks Responsiveness for Background Tasks**
   (a) Some operations like file upload/download run synchronously, causing the UI to freeze momentarily during heavy tasks.
   (b) No use of threading or async functions in the UI layer for smooth user experience.

9. **No Metadata Editing or Visualization**
   (a) Although metadata is logged in `file_metadata.json`, users cannot view, edit, or manage this metadata from the UI.

10. **Limited Fault Recovery Logic**
    (a) If both the primary and replica storage nodes fail or become unreachable, file recovery is not possible.

(b) The system does not attempt partial file reconstruction or notify users of missing parts with diagnostic suggestions.

11. **No Version Control**

    (a) Uploading a file with the same name overwrites existing metadata without maintaining version history.

    (b) Versioning or rollback functionality is absent.

12. **Lack of Automated Backup or Archival**

    (a) There is no option to back up file metadata or server configurations automatically.

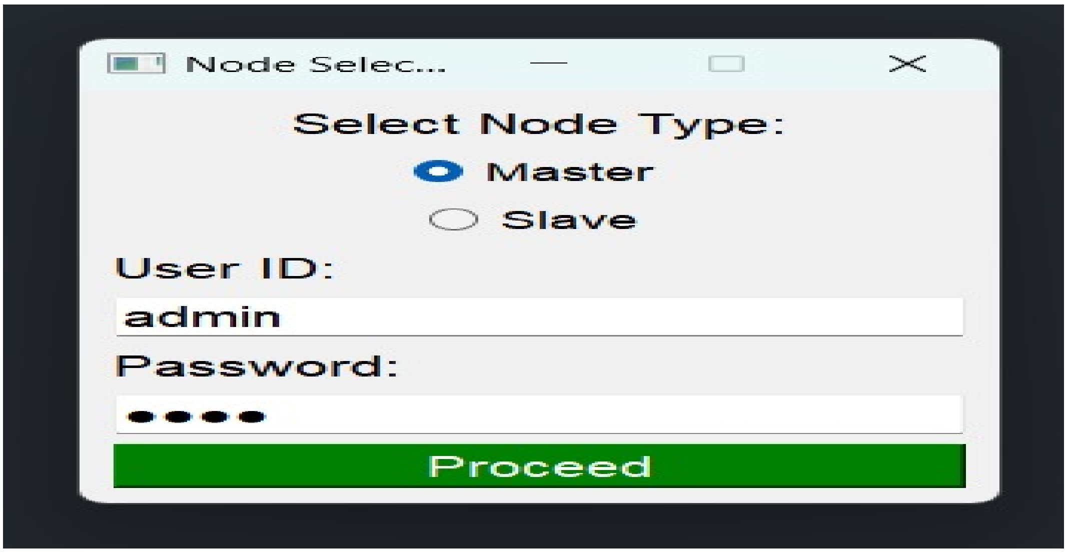
    (b) Users must manually manage all logs and configurations.

9 TESTING & OUTPUT

1. **Unit Testing:** Verifying individual modules.
2. **Integration Testing:** Ensuring different components work together.
3. **Performance Testing:** Simulating large-scale data loads.
4. **Security Testing:** Checking for vulnerabilities in access control.

Expected Output:

1. Efficient file storage and retrieval.
2. Fast and secure access through RBAC.
3. Optimized performance with sharding and load balancing.



Select Node Type:

● Master

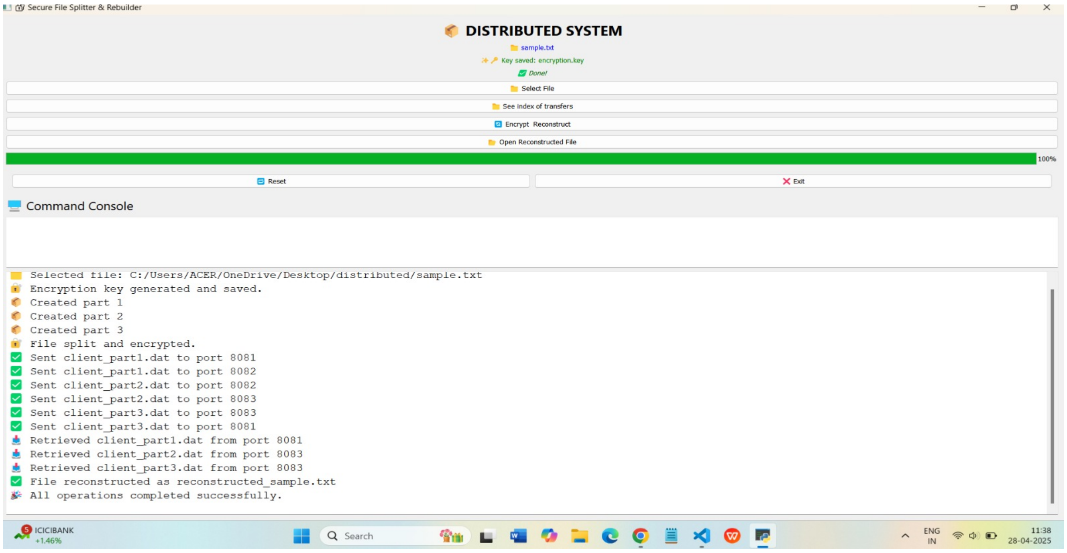
○ Slave

User ID: admin

Password: ●●●●

Figure 8: Login window

Figure 8 shows the initial user interface of the application. This gives us two options whether to login as master or slave providing us with the respective previleges



sample.txt

Key saved: encryption.key

Done!

```
Selected file: C:/Users/ACER/OneDrive/Desktop/distributed/sample.txt
Encryption key generated and saved.
Created part 1
Created part 2
Created part 3
File split and encrypted.
Sent client_part1.dat to port 8081
Sent client_part1.dat to port 8082
Sent client_part2.dat to port 8082
Sent client_part2.dat to port 8083
Sent client_part3.dat to port 8083
Sent client_part3.dat to port 8081
Retrieved client_part1.dat from port 8081
Retrieved client_part2.dat from port 8083
Retrieved client_part3.dat from port 8083
File reconstructed as reconstructed_sample.txt
All operations completed successfully.
```

Figure 9: User interface

Figure ?? shows the master user interface of the application. The interface is designed to be clean and intuitive, enabling users to easily navigate through the platform. It includes clearly labeled buttons, input fields, and responsive elements that enhance the user experience.

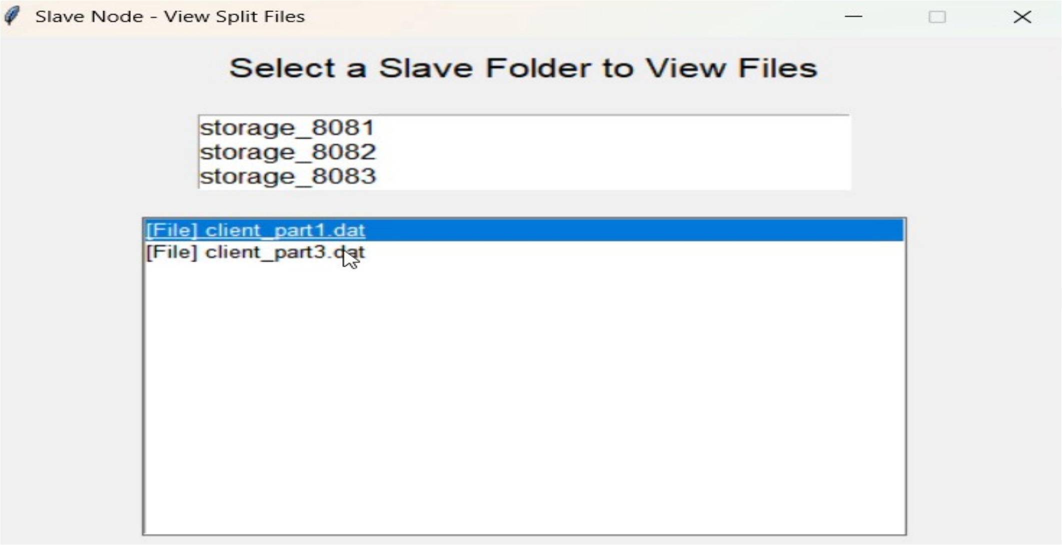
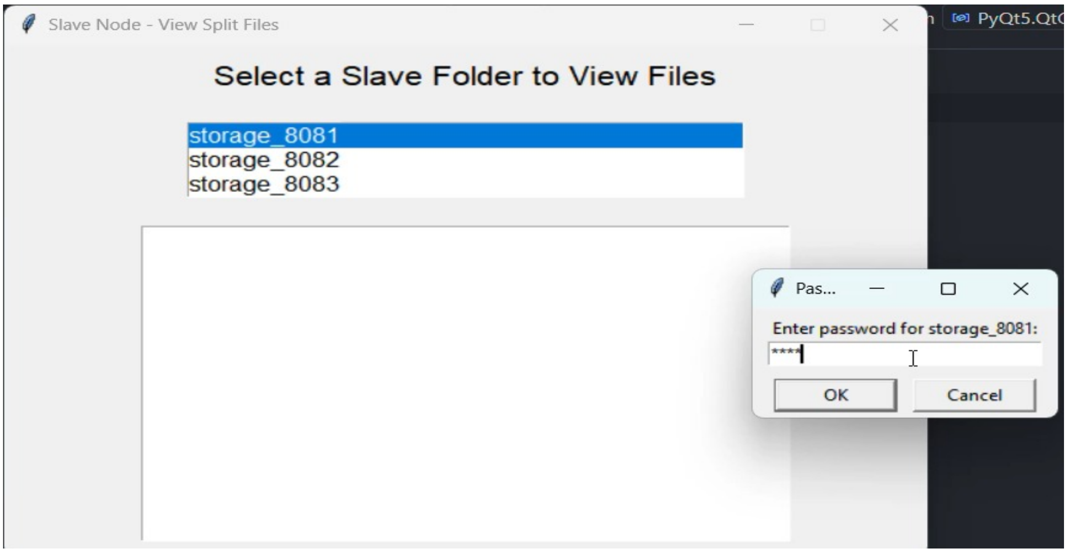


Figure 10: Slave view

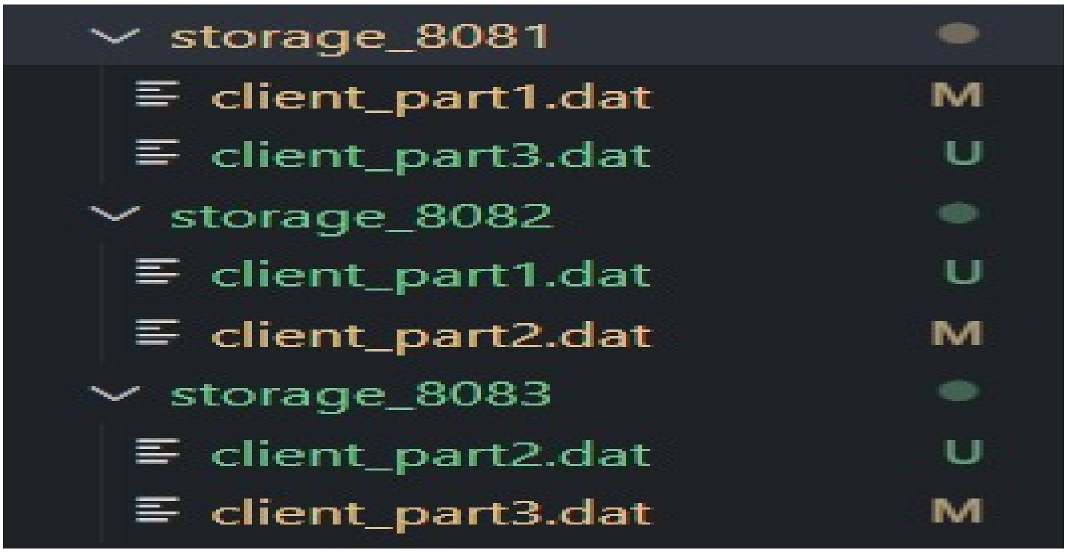
Figure ?? shows the slave user interface of the application. This only permits the slave to just view the files whether they are just availabe in a particular file.


Slave Node - View Split Files
Select a Slave Folder to View Files
storage_8081
storage_8082
storage_8083
PyQt5.Qt
Pas...
OK
Cancel

Enter password for storage_8081: ****

Figure 11: Slave view

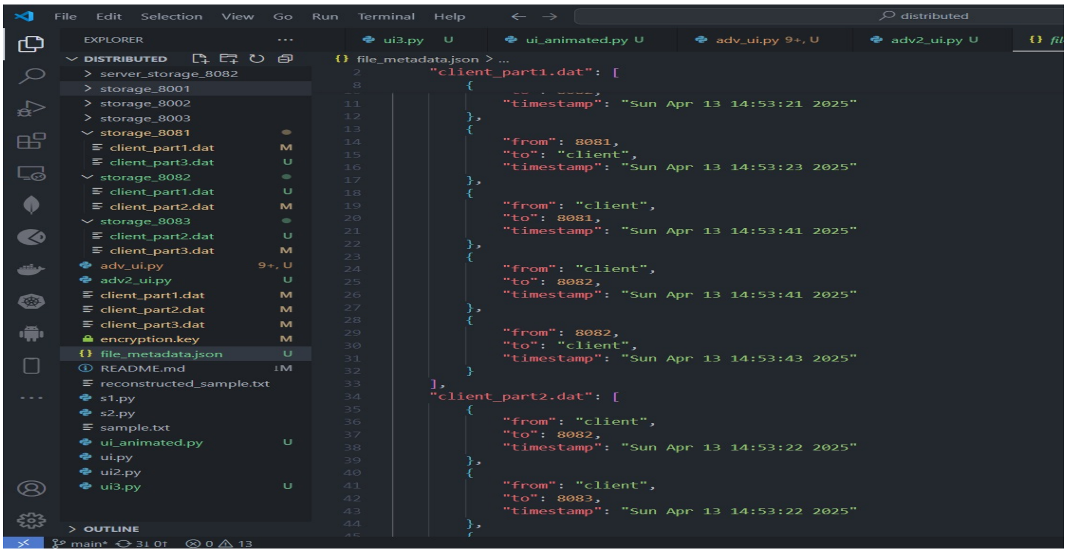
Figure ?? This only permits the particular slave to just view the files whether they are just availabe in a particular file.



```
∨ storage_8081                ●
    client_part1.dat          M
    client_part3.dat          U
∨ storage_8082                ●
    client_part1.dat          U
    client_part2.dat          M
∨ storage_8083                ●
    client_part2.dat          U
    client_part3.dat          M
```

Figure 12: Rack awareness

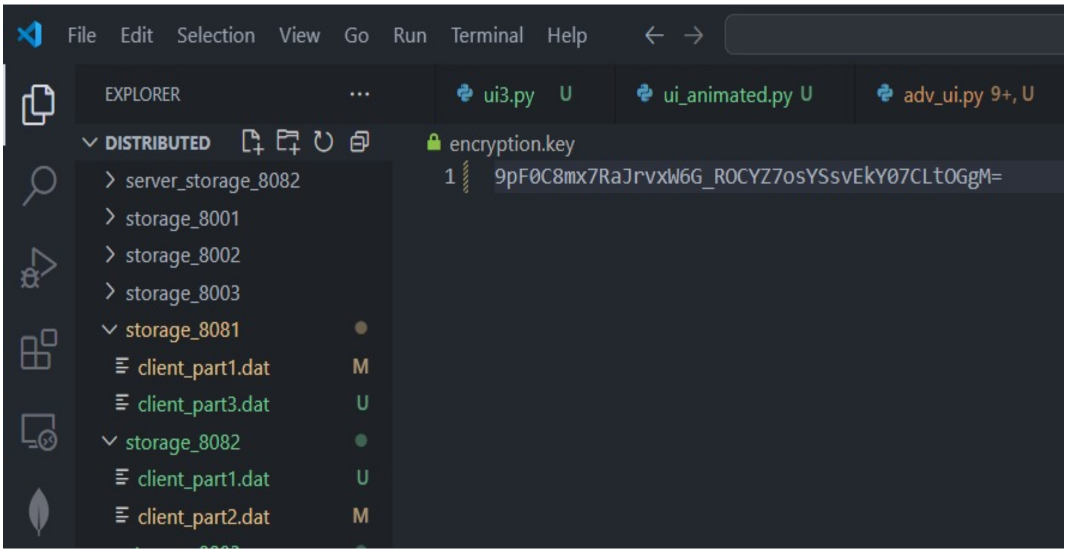
Figure 12 illustrates the concept of rack awareness in a distributed system. Rack awareness helps the system place replicas of data blocks in different racks to ensure fault tolerance and data availability even if an entire rack fails. This mechanism enhances data reliability and efficient resource utilization.



```
    "client_part1.dat": [
        {
            "timestamp": "Sun Apr 13 14:53:21 2025"
        },
        {
            "from": 8081,
            "to": "client",
            "timestamp": "Sun Apr 13 14:53:23 2025"
        },
        {
            "from": "client",
            "to": 8081,
            "timestamp": "Sun Apr 13 14:53:41 2025"
        },
        {
            "from": "client",
            "to": 8082,
            "timestamp": "Sun Apr 13 14:53:41 2025"
        },
        {
            "from": 8082,
            "to": "client",
            "timestamp": "Sun Apr 13 14:53:43 2025"
        }
    ],
    "client_part2.dat": [
        {
            "from": "client",
            "to": 8082,
            "timestamp": "Sun Apr 13 14:53:22 2025"
        },
        {
            "from": "client",
            "to": 8083,
            "timestamp": "Sun Apr 13 14:53:22 2025"
        },
```

Figure 13: Metadata

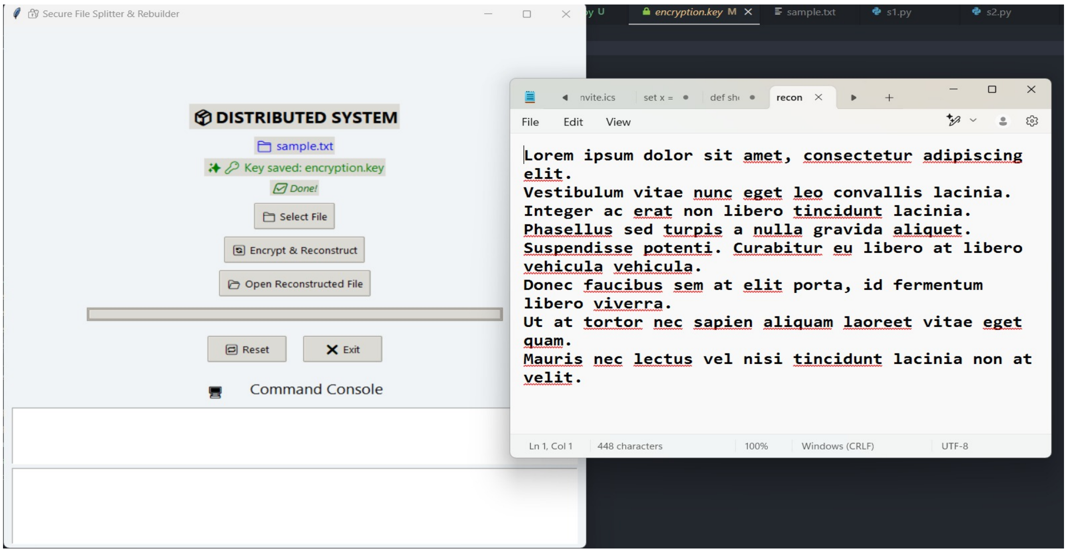
Figure 13 shows how metadata is managed in the system. Metadata includes information such as file names, block locations, permissions, and timestamps. Proper management of metadata is critical for quick data retrieval and maintaining the integrity of the file system.



```
9pF0C8mx7RaJrvxW6G_ROCYZ7osYSsvEkY07CLtOGgM=
```

Figure 14: Encryption key

Figure 14 displays the encryption mechanism using keys to secure data. Encryption ensures that even if data is intercepted or accessed without permission, it cannot be read without the appropriate decryption key. This is vital for maintaining data privacy and security.



```
Lorem ipsum dolor sit amet, consectetur adipiscing
elit.
Vestibulum vitae nunc eget leo convallis lacinia.
Integer ac erat non libero tincidunt lacinia.
Phasellus sed turpis a nulla gravida aliquet.
Suspendisse potenti. Curabitur eu libero at libero
vehicula vehicula.
Donec faucibus sem at elit porta, id fermentum
libero viverra.
Ut at tortor nec sapien aliquam laoreet vitae eget
quam.
Mauris nec lectus vel nisi tincidunt lacinia non at
velit.
```

Figure 15: Merged file

Figure 15 represents a merged file scenario, where multiple file segments or logs are combined into a single coherent file. This process improves performance by reducing fragmentation and simplifying data management, especially in systems that generate many small files.

10 FUTURE WORKS

As the distributed file system evolves, several future enhancements have been identified to extend its functionality, improve performance, and address emerging challenges:

10.1 Enhanced Security Features

1. **Granular Access Control:**
   (a) Introducing Role-Based Access Control (RBAC) to define user roles and enforce data access policies.
   (b) Protection against unauthorized interactions and secure data segregation.
2. **Encryption Key Management:**
   (a) Implementing secure key management practices including automated key rotation.
   (b) User-specific encryption keys for enhanced data confidentiality.

10.2 Intelligent Querying and Data Operations

1. **AI-Powered File Search:**
   (a) Integration of AI-based indexing and semantic search mechanisms.
   (b) Faster and more intuitive file retrieval for large-scale databases.
2. **Metadata Tagging:**
   (a) Expansion of metadata system to support custom tags and timestamps.
   (b) Enhanced versioning and file lifecycle management capabilities.

10.3 Scalability and Performance Improvements

1. **Dynamic Load Balancing:**
   (a) Replacement of round-robin with adaptive load balancing.
   (b) Real-time server health monitoring and traffic rerouting.
2. **Data Replication Policies:**
   (a) Configurable replication strategies including geo-redundancy.
   (b) Improved redundancy management for distributed environments.

10.4 User Interface and Usability Enhancements

1. **Real-Time Monitoring Dashboard:**
   (a) Development of live dashboard for system visualization.
   (b) Monitoring of file transfers, server statuses, and diagnostic logs.
2. **Cross-Platform Support:**
   (a) Porting client GUI to web and mobile platforms.
   (b) Enhanced accessibility across different environments.

10.5 Community and Ecosystem Development

1. **Plugin System for Extensibility:**
   (a) Framework for developer-created plugins (compression, antivirus, etc.).
   (b) Increased adaptability to diverse use cases.
2. **Open-Source Collaboration:**
   (a) Publication as open-source project to foster community contributions.
   (b) Collaborative innovation through bug fixes and feature suggestions.
3. **User Feedback Integration:**
   (a) Built-in feedback channels for real-time user insights.
   (b) Continuous improvement through iterative development.

11 CONCLUSION

This project successfully demonstrates a scalable and secure distributed file management system using a client-server architecture with redundancy and encryption. The client application offers an intuitive graphical interface for selecting files, which are then encrypted, split into multiple parts, and distributed to multiple server instances. The system leverages load balancing strategies and fault tolerance by maintaining both main and replica storage endpoints, ensuring reliable file transmission and recovery even in cases of server failure.

Security is enforced through symmetric encryption using the Fernet cipher, protecting sensitive data during storage and transmission. Metadata tracking further enhances traceability and auditability. Servers are designed to operate concurrently and independently, enabling horizontal scalability and robustness.

While the current system focuses on encrypted file distribution and reconstruction with basic fault handling and visual feedback, future enhancements may include fine-grained Role-Based Access Control (RBAC), AI-driven content indexing and search, real-time monitoring dashboards, and dynamic server scaling based on workload.

12 REFERENCES

1. Kleppmann, Martin. Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems. " O'Reilly Media, Inc.", 2017. `http://chrome-extension/efaidnbmnnnibpcajpcglclefindmkaj/https:/www.scylladb.com/wp-content/uploads/ScyllaDB-Designing-pdf`.

2. Yang, Chen. *Theoretical Analysis of Distributed Systems and Their Scalability.* Clausius Scientific Press, 2025. `https://www.clausiuspress.com/assets/default/article/2025/02/24/article_1740379993.pdf`

3. **Depardon, Benjamin, Gaël Le Mahec, and Cyril Séguin.** "Analysis of Six Distributed File Systems." INRIA Research Report (2013). `https://inria.hal.science/hal-00789086/file/a_survey_of_dfs.pdf`

4. **Golab, Wojciech, and Mohammad R. Rahman.** "Consistency Models in Distributed Systems: A Survey on Definitions, Disciplines, Challenges and Applications." arXiv preprint arXiv:1902.03305 (2019). `https://arxiv.org/abs/1902.03305`

5. Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. "The Google file system." In Proceedings of the nineteenth ACM symposium on Operating systems principles, pp. 29-43. 2003. `https://dl.acm.org/doi/abs/10.1145/945445.945450`

6. Stonebraker, Michael, and Andrew Pavlo. "What Goes Around Comes Around... And Around..." ACM Sigmod Record 53, no. 2 (2024): 21-37. `https://dl.acm.org/doi/abs/10.1145/3685980.3685984`

7. Simon, Salomé. "Brewer's cap theorem." CS341 Distributed Information Systems, University of Basel (HS2012) (2000). `chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https:/static.googleusercontent.com/media/research.google.com/en/pubs/archive/45855.pdf`